Scheduling linear chain streaming applications on heterogeneous systems with failures

Anne Benoit, Alexandru Dobrila, Jean-Marc Nicod, Laurent Philippe

Scheduling linear chain streaming applications on heterogeneous systems with failures

Anne Benoit[a], Alexandru Dobrila[b], Jean-Marc Nicod[b], Laurent Philippe[b]

[a] *ENS Lyon, Université de Lyon, LIP laboratory (ENS, CNRS, INRIA, UCBL), France*
[b] *FEMTO-ST Institute, UMR CNRS / UFC / ENSMM / UTBM, Besançon, France*

Abstract

In this paper, we study the problem of optimizing the throughput of streaming applications for heterogeneous platforms subject to failures. Applications are linear graphs of tasks (pipelines), with a type associated to each task. The challenge is to map each task onto one machine of a target platform, each machine having to be specialized to process only one task type, given that every machine is able to process all the types before being specialized in order to avoid costly setups. The objective is to maximize the throughput, i.e., the rate at which jobs can be processed when accounting for failures. Each instance can thus be performed by any machine specialized in its type and the workload of the system can be shared among a set of specialized machines.

For identical machines, we prove that an optimal solution can be computed in polynomial time. However the problem becomes NP-hard when two machines may compute the same task type at different speeds. Several polynomial time heuristics are designed for the most realistic specialized settings. Simulation results assess their efficiency, showing that the best heuristics obtain a good throughput, much better than the throughput obtained with a random mapping. Moreover, the throughput is close to the optimal solution in the particular cases where the optimal throughput can be computed.

Keywords: heterogeneous computing, scheduling, throughput maximization, failure, streaming applications, complexity results, linear programming.

1. Introduction

In this paper, we address the issue of mapping a linear chain of tasks that processes a flow of jobs on heterogeneous resources subject to failures. Note that the work can be extended to the more general case when the throughput is not preserved all along the chain, i.e., the output throughput of the tasks of

Email addresses: Anne.Benoit@ens-lyon.fr (Anne Benoit), alex.dobrilla@gmail.com (Alexandru Dobrila), Jean-Marc.Nicod@femto-st.fr (Jean-Marc Nicod), Laurent.Philippe@femto-st.fr (Laurent Philippe)

the pipeline may be smaller than their input throughput. This may arise for instance in streaming applications, either because the task operates some kind of selection on the input data, or when the task is not able to compute the output, for instance because of failures. So we rather consider the issue of task failures than machine failures.

A streaming application is composed of a flow of elementary jobs (job instances of the same size). Each of these elementary jobs is in turn composed of tasks, linked by precedence constraints. Thus, the platform must continuously execute instances of elementary jobs. The objective is to map the tasks onto a computational platform, consisting of several resources, in order to optimize the job flow through the platform. The goal is therefore to maximize the number of job output per time unit (the throughput), or equivalently, to minimize the time between two output jobs (the period). The problem is rather simple when the resources are homogeneous, but becomes more complex when considering heterogeneous platforms. The originality of our work is that we assume that the flow reduction may be linked to the tasks and/or to the processing resources.

The paper is organized as follows. We first define more precisely the context and we give an overview of related work in Section 2. Then we present the framework, define the failure model and formalize the optimization problems in Section 3. An exhaustive study on the complexity of these problems is provided in Section 4: we exhibit some particular polynomial problem instances, we prove that the remaining problem instances are NP-hard, and we propose some linear programming formulations to solve sub-problems. In Section 5, we design a set of polynomial-time heuristics to solve the most general problem instance. In Section 6, we conduct extensive simulations to assess the relative and absolute performance of the heuristics. Finally, we conclude in Section 7.

2. Context and related work

In the past years, much attention has been paid to workflow applications on the grid and several workflow management systems facilitate their execution on the computing resources [27, 28]. We focus in this paper on linear chain streaming applications, executed on computing grids. In this case, the resources are distributed, heterogeneous, and may not be reliable enough to assume that the failure rate can be ignored in the mapping strategy.

From the application point of view, we deal with *coarse-grain* streaming applications. Applications are a linear graph of tasks (pipeline) with a type associated to each task. In these streaming applications, a series of data enters the pipeline and progresses from task to task until the final result is computed. Examples of such applications are stream-processing applications composed of processing elements as in [20], or pipelined query operators with precedence constraints as in [7], or application based on stream programming as in [13]. An illustrating example of a streaming application is an image processing application as presented in [15]. A stream of intra-vascular ultrasound images are captured by a transducer at a specific rate and sent to the resources to be processed before being displayed (see Figure 1). A similar application is the

Synthetic Aperture Radar (SAR) [16], which creates 2D or 3D images from radar signals gathered by a moving sensor. These images are then used to make decisions. It is important to note that for such streaming applications, it is more relevant to optimize the throughput rather than the total finish time.

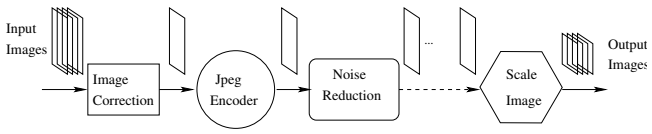

Figure 1: Application example.

The considered resources are typically dedicated execution resources grouped in a distributed platform, a grid, on which we process either a batch of input data. Each resource of the platform provides functions or services that are able to handle a task type. As each task is typed it can only be processed on a resource that implements its task type. In the case of computing resources this model can be illustrated by Software as a Service (SaaS) based platforms [1]. The considered platforms are heterogeneous as the resources are usually not uniform and thus the tasks are processed with different speeds.

Our aim is to efficiently map and schedule the applications onto the resources. We target coarse-grain applications and platforms such that the cost of communications is negligible in comparison to the cost of computations. In the illustrating example, processing an image is indeed much more costly than transferring an image. This is a complex problem (known in the literature as *multi-processor tasks* [6, 14]) as the considered resources are heterogeneous. The mapping defines which resource performs which task. So processing a streaming application on the platform amounts to enter jobs on the platform and to progress from resource to resource, following the task chain, until the final result is computed. After an initialization delay, a new job is completed every period and it exits the pipeline. The period is therefore defined as the longest cycle-time of a resource, and it is the inverse of the throughput that can be achieved. The goal is to minimize the period, which is equivalent to maximizing the throughput, i.e., the number of final results that exit the system per time unit. This approach is different from [7, 11] where pipelined query operators with precedence constraints are ordered to optimize a bottleneck metric, the slowest stage in the pipeline and the selectivity of the operators. In our case, the operation order is fixed and we target the operator mapping on the resources.

Note that optimizing the schedule of a set of tasks on a heterogeneous platform is complex (NP-hard most of the time), as we will show later in the paper. Given that the optimization target in the throughput optimization and that the tasks of a streaming application remain identical all along the execution, it is worth to take the time to compute a static assignment before execution since the execution parameters, and in particular the execution time of the tasks, do not change during the streaming execution.

Considering platforms such as grids, or clouds, implies to take failure possibilities into account. Failures cannot be ignored when applications last for a long time. The failure rate is too high to assume that no fault will impact the execution. In the grid context, failures may occur because of the nodes, but they also may be related to the complexity of the service [18]. So we consider in our problem that the type of a task affects its computation requirements and its failure rate. This failure rate may also depend on the resource itself, platform heterogeneity also assumes reliability heterogeneity.

Replication is often used to deal with failures in distributed systems [8, 23]. To ensure that a result is output the same execution is replicated among several processors. However, a common property of our target platforms is that we cannot use replication to overcome the faults. For streaming applications, it is too costly to replicate each task (maybe several times if we want a high warranty) and replication shrinks the throughput. Fortunately, losing a few jobs may not be a big deal; for instance, the loss of some images in the illustrating example will not alter the result, as far as the throughput is maintained. This failure model is based on the Window-Constrained [24] model, often used in real-time environment. In this model, only a fraction of the messages will reach their destination: for y messages, only x ($x \leq y$) of them will reach their destination. The y value is called the Window. The losses are not considered as a failure but as a guarantee: for a given network, a Window-Constrained scheduling [26, 25] can guarantee that no more than x messages will be lost for every y sent messages.

Other research work already focuses on streaming applications. It operates on data sets but most of the time on homogeneous resources [19]. Other studies as [22, 10] target workflow application scheduling on grids but more from a practical point of view. A comprehensive survey of pipelined workflow scheduling is given in [2] but it does not tackle the fault tolerance issue. In [4] a computation is considered to be faulty in case where data is lost. Replication is used to improve the reliability of the system and to optimize two objective functions, the latency and the reliability. This is different from the case tackled in this paper as no throughput change along the pipeline is considered.

In this paper, we therefore solely concentrate on the problem of period minimization (i.e., throughput maximization), where extra jobs are processed to account for failures. For instance, if there is a single task, mapped on a single machine, with a failure rate of $1/2$, a throughput of x jobs per unit time will be achieved if the task processes $2 \times x$ jobs per time unit.

3. Framework and optimization problems

In this section, we define the problems that we tackle. First we present the application, platform and failure models. Then we discuss the objective function and the rules of the game before formally introducing the optimization problems.

3.1. Applicative framework

The application consists of a linear chain of n tasks, $T_1, T_2, \ldots, T_n$ as presented on Figure 2. A type is associated to each task as the same operation may be applied several times to the same job. We have a set $\mathcal{T}$ of p task types with $n \geq p$, and a function $t : \{1, \ldots, n\} \to \mathcal{T}$, which returns the task type from the task number. Hence, $t(i)$ is the type of task T_i.

A series of jobs enters the workflow and progresses from task to task until the final result is computed. We note x_i the average number of jobs processed by task T_i to output one job out of the system. Note that x_{i+1} depends on x_i and on the failure rate of the machine processing T_i (see below).

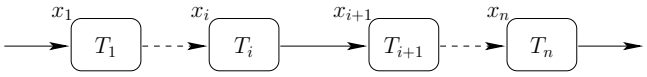

Figure 2: Application model

3.2. Target platform

The target platform is distributed and heterogeneous. It consists of a set $\mathcal{M}$ of m machines (a cell in the micro-factory or a host in a grid platform): $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$.

The task processing time depends on the machine that performs it. It takes $w_{i,u}$ units of time to machine M_u to execute task T_i on one job. All the tasks of the same type are processed in the same time on a given machine, i.e., if $t(i) = t(j)$, then $w_{i,u} = w_{j,u}$ (for $1 \leq u \leq m$). Moreover, each machine is able to process all the task types. But, to avoid costly context or setup changes during execution, the machines may be specialized to process only one task type. Moreover, the machines are interconnected by a complete graph but we do not take communication times into account as we consider that the processing time is much greater than the communication time (coarse-grain applications).

Our goal is to assign tasks to machines so as to optimize some key performance criteria. In some of the considered cases a task can be allocated to several machines. We define $q(i, u)$ as the quantity of task T_i executed by machine M_u; if $q(i, u) = 0$, T_i is not assigned to M_u. Note that $q(i, u)$ defines an allocation function and for each task T_i we must have at least one $q(i, u) > 0$, meaning that T_i is allocated to M_u.

Recall that x_i is the average number of jobs processed by task T_i to output one job out of the system. We must have, for each task T_i, $\sum_{u=1}^{m} q(i, u) = x_i$, i.e., enough jobs are processed for task T_i in the system.

3.3. Failure model

An additional characteristic of our framework is that failures may occur. It may happen that a job (or data, or product) is lost (or damaged) while a task is being executed on this job instance. For instance, a message will be lost due to network contention. Note that we only deal with transient failures, as defined

in [17]: the tasks are failing for some of the job instances. Therefore, we process more jobs than needed, so that at the end, the required throughput is reached. The failure rate of task T_i performed onto machine M_u is the percentage of failure for this task and it is denoted $f_{i,u}$. Therefore, if all jobs for task T_i are executed on machine M_u, we have $x_{i+1} = x_i(1 - f_{i,u})$.

Note that we do not consider permanent failures in this paper. Our goal is to statically assign tasks to machines and, in that case, a permanent failure of one machine implies to reconsider the whole assignment as this would lead to a failure for all the remaining jobs to be processed and the inability to finish them. This issue should rather be addressed by replication and has already been studied, for instance in [5]. However, since the heuristics proposed in this paper have a polynomial time complexity, they could be dynamically used to compute a new assignment for the tasks in the case of a permanent failure.

3.4. Objective function

In our framework, several objective functions could be optimized. For instance, one may want to produce a mapping of the tasks on the machines as reliable as possible, i.e., minimize the number of products to input in the system. Rather, we consider that losing one instance is not a big deal, and we focus on a performance criteria, the throughput. The goal is to maximize the number of instances that exit the system per time unit, making abstraction of the initialization and clean-up phases. This objective is important when a large number of instances must be processed. Note that we deal with the equivalent optimization problem that minimizes the *period*, the inverse of the throughput.

One challenge is that we cannot compute the number x_i of jobs that must be processed by task T_i before allocating tasks to machines, since x_i depends on the failure rates incurred by the allocation. However, each task T_i has a unique successor task T_{i+1}, and x_{i+1} is the amount of jobs needed by T_{i+1} as input. Since T_i is distributed on several machines with different failure rates, we have $\sum_{u=1}^{m} (q(i,u) \times (1 - f_{i,u})) = x_{i+1}$, where $q(i,u) \times (1 - f_{i,u})$ represents the amount of jobs output by the machine M_u if $q(i,u)$ jobs are treated by that machine. For each task, we sum all the instances treated by all the machines.

We are now ready to define the cycle-time ct_u of machine M_u: it is the time needed by M_u to execute all tasks T_i with $q(i,u) > 0$: $ct_u = \sum_{i=1}^{n} q(i,u) \times w_{i,u}$. The objective function is to minimize the maximum cycle-time, which corresponds to the period of the system: $\min(\max_{1 \leq u \leq m}(ct_u))$.

3.5. Rules of the game

Different rules of the game may be enforced to define the allocation, i.e., the $q(i,u)$ values. For *one-to-many* mappings, we enforce that one task can be mapped onto several machines but one machine can only perform one task (or a fraction of one task): $\forall 1 \leq i, i' \leq n \; s.t. \; i \neq i', \; q(i,u) > 0 \Rightarrow q(i',u) = 0$. For instance, on Figure 3, we have an application graph with three tasks (a) that are mapped onto the platform graph with four resources (b). The result is shown in (c) where we can see that one machine can handle only one task: M_1 and

M_2 are both processing T_2, while M_3 processes T_1 and M_4 processes T_3. This mapping is quite restrictive because we must have at least as many machines as tasks. Note that a task can be split into several instances, as for T_2 in the example: part of the jobs are processed by M_1, and the remaining jobs are processed by M_2.

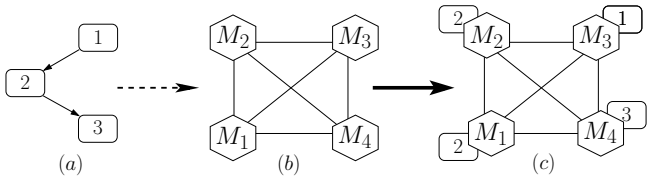

Figure 3: *One-to-many* mapping.

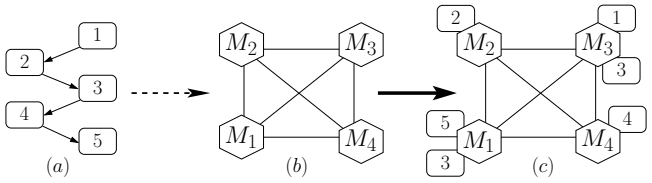

Figure 4: *Specialized* mapping. t(1)=t(3)=t(5)=1 and t(2)=t(4)=2.

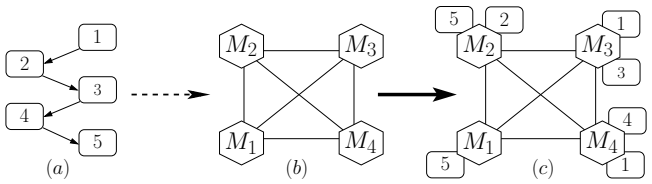

Figure 5: *General* mapping. t(1)=t(3)=1, t(2)=t(4)=2 and t(5) = 3.

We relax this rule to define *specialized* mappings, in which several tasks of the same type can be mapped onto the same machine: $\forall 1 \leq i, i' \leq n \; s.t. \; t(i) \neq t(i'), \; q(i,u) > 0 \Rightarrow q(i',u) = 0$. For instance, on Figure 4, we have five tasks with types $t(1) = t(3) = t(5) = 1$ and $t(2) = t(4) = 2$. Machine M_3 computes task T_1, therefore it can also execute T_3 and T_5 but not T_2 and T_4. As types are not dedicated to machines, T_5 can also be assigned to another machine, for instance M_1. Note that if each task has a different type, the specialized mapping and the one-to-many mapping are equivalent.

Finally, *general* mappings have no constraints: any task (no matter the type) can be mapped on any machine, as illustrated on Figure 5. Note that general mappings can also be seen as a special case of specialized mappings, by assuming that all tasks have the same type.

3.6. Problem definition

We are now ready to formally define the optimization problems tackled in this paper. The three important parameters are:

- the rules of the game: *one-to-many (o2m)*, *specialized (spe)* or *general (gen)* mapping;
- the failure model: f if failures are all identical on all the machines, f_i if the failure for a same task is identical on two different machines, f_u if the failure rate depends only on the machine, and the general case $f_{i,u}$;
- the computing time: w if the processing times are all identical, w_i if it differs only from one task to another, w_u if it depends only on the machine, and $w_{i,u}$ in the general case.

Definition of $\textsc{MinPer}(R, F, W)$**.** *Given an application and a target platform, with a failure model* $F = \{f|f_i|f_u|f_{i,u}|*\}$ *and computation times* $W = \{w|w_i|w_u|w_{i,u}|*\}$, $\textsc{MinPer}(R, F, W)$ *finds a mapping, i.e., values of* $q(i, u)$ *such that for each task* T_i $(1 \leq i \leq n)$, $\sum_{u=1}^{m} q(i, u) = x_i$, *following rule* $R = \{o2m|spe|gen|*\}$, *which minimizes the period of the application,* $\max_{1 \leq u \leq m} \sum_{i=1}^{n} q(i, u) \times w_{i,u}$.

For instance, $\textsc{MinPer}(o2m, f_i, w_i)$ is the problem of minimizing the period with a *one-to-many* mapping, where both failure rates and execution times depend only on the tasks. Note that $*$ is used to express the problem with any variant of the corresponding parameter; for instance, $\textsc{MinPer}(*, f_{i,u}, w)$ is the problem of minimizing the period with any mapping rule, where failure rates are general, while execution times are all identical.

4. Complexity results

We assess the complexity of the different instances of the $\textsc{MinPer}(R, F, W)$ problem. First we provide the complexity of the problems with $F = f_i$ (Section 4.1), and then we discuss the most general problems with $F = f_{i,u}$ (Section 4.2). We do not address problems with $F = f_u$ in this paper as, in the context of this study, the complexity of the considered tasks is the main failure factor. Some results on these $F = f_u$ problems are presented in [3].

Even though the general problems are NP-hard, we show that once the allocation of tasks to machines is known, we can optimally decide how to share tasks between machines, in polynomial time (Section 4.3). Also, we give an integer linear program to solve the problem (Section 4.4).

4.1. Complexity of the $\textsc{MinPer}(*, f_i, *)$ *problems*

In this section, we focus on the $\textsc{MinPer}(*, f_i, *)$ problems, and we first show how these problems can be simplified. Indeed, in this case, the number of products that should be computed for task T_i at each period, x_i, is independent of the allocation of tasks to machines. We can therefore ignore the failure probabilities, and focus on the computation of the period of the application.

The following Lemma 1 allows us to further simplify the problem for the most difficult instance of specialized mappings: tasks of similar type can be grouped and processed as a single *equivalent* task.

Lemma 1. *For* MinPer(spe, f_i, w_i) *or* MinPer(spe, f_i, w_u)*, there exists an optimal solution in which all tasks of the same type are executed onto the same set of machines, in equal proportions. Given the optimal mapping and its allocation function q, we have:*

$$\begin{gathered} \forall 1 \leq i, j \leq n \text{ with } t(i) = t(j), \\ \exists\, \alpha_{i,j} \in \mathbb{Q} \text{ such that } \forall 1 \leq u \leq m,\ q(i,u) = \alpha_{i,j} \times q(j,u)\ . \end{gathered} \tag{1}$$

Proof. Let OPT be an optimal solution to the problem, of period P. Let t be a task type, and, without loss of generality, let $T_1, \ldots, T_k$ be the k tasks of type t, with $k \leq n$, and let $M_1, \ldots, M_v$ be the set of machines specialized to type t in the optimal solution OPT (i.e., they have been allocated only tasks from $T_1, \ldots, T_k$), with $v \leq m$. In the optimal solution, $q(i,u)$ is the proportion of task T_i assigned to machine M_u. For each task T_i, $1 \leq i \leq k$, we have $\sum_{1 \leq u \leq v} q(i,u) = x_i$, and for each machine M_u, $1 \leq u \leq v$, we have $\sum_{1 \leq i \leq k} q(i,u) w_{i,u} \leq P$, where $w_{i,u}$ may be either w_i or w_u, depending upon the problem instance.

We build a new optimal solution, OPT', which follows Equation (1). The proportion of task T_i assigned to machine M_u in this solution is $q'(i,u)$, and $x^* = \sum_{1 \leq i \leq k} x_i$.

Let us start with the MinPer(spe, f_i, w_u) problem. We define $q'(i,u) = \frac{x_i}{x^*} \times \frac{P}{w_u}$. For task T_i, $\sum_{1 \leq u \leq v} q'(i,u) \geq \frac{x_i}{x^*} \times \sum_{1 \leq u \leq v} \sum_{1 \leq i \leq k} q(i,u)$, by definition of OPT, and therefore $\sum_{1 \leq u \leq v} q'(i,u) \geq x_i$: we have distributed all the work for this task. Moreover, by construction, $\sum_{1 \leq i \leq k} q'(i,u) w_u = P$: solution OPT' is optimal (its period is P). We have built an optimal solution that satisfies Equation (1), with $\alpha_{i,j} = \frac{x_i}{x_j}$, by redistributing the work of each task on each machine in equal proportion.

The reasoning is similar for the MinPer(spe, f_i, w_i) problem, except that w_i is now depending on the task, and therefore we define $q'(i,u) = \frac{x_i}{w_i x^*} \times P$. We still have the property that all the work is distributed, and the period of each machine is still P. The only difference relies in the values of $\alpha_{i,j}$, which are now also depending upon the w_i: $\alpha_{i,j} = \frac{x_i}{x_j} \times \frac{w_j}{w_i}$.

For each problem instance, we have built an optimal solution that follows Equation (1), therefore concluding the proof. □

Corollary 1. *For* MinPer(spe, f_i, w_i) *or* MinPer(spe, f_i, w_u)*, we can group all tasks of same type t as a single* equivalent *task $T_t^{(eq)}$ such that*

$$x_t^{(eq)} = \sum_{1 \leq i \leq n | t(i) = t} x_i\ .$$

Then, we can solve this problem with the one-to-many rule, and deduce the solution of the initial problem.

Proof. Following Lemma 1, we search for the optimal solution which follows Equation (1). Since all tasks of the same type are executed onto the same set of machines in equal proportions, we can group them as a single equivalent task.

The amount of work to be done by the set of machines corresponds to the total amount of work of the initial tasks, i.e., for a type t, $\sum_{1 \leq i \leq n | t(i)=t} x_i$.

The one-to-many rule decides on which set of machines each equivalent task is mapped, and then we share the initial tasks in equal proportions to obtain the solution to the initial problem: if task T_i is not mapped on machine M_u, then $q(i,u) = 0$, otherwise

$$q(i,u) = \frac{x_i}{x_{t(i)}^{(eq)}} \times \frac{P}{w_{i|u}} ,$$

where $w_{i|u} = \{w_i \mid w_u\}$, depending upon the problem instance. □

We are now ready to establish the complexity of the MinPer$(*, f_i, *)$ problems. Recall that n is the number of tasks, m is the number of machines, and p is the number of types.

We start by providing polynomial algorithms for one-to-many and specialized mappings with w_i (Theorem 1 and Corollary 2). Then, we discuss the case of general mappings, which can also be solved in polynomial time (Theorem 2). Finally, we tackle the instances that are NP-hard (Theorem 3).

Theorem 1. MinPer$(o2m, f_i, w_i)$ *can be solved in polynomial time* $O(m \times \log n)$.

Proof. First, note that solving this one-to-many problem amounts to decide on how many machines each task is executed (since machines are identical), and then split the work evenly between these machines to minimize the period. Hence, if T_i is executed on k machines, $q(i,u) = \frac{x_i}{k}$, where M_u is one of these k machines, and the corresponding period is $\frac{x_i}{k} \times w_i$.

We exhibit a dynamic programming algorithm that computes the optimal solution in polynomial time. We compute $P(i,k)$, which is the period that can be achieved to process tasks $T_1, \ldots, T_i$ with k machines. The solution to the problem is $P(n,m)$, and the recurrence writes:

$$P(i,k) = \min_{1 \leq k' \leq k} \left(\max \left(P(i-1, k-k'), \frac{x_i}{k'} \times w_i \right) \right) ,$$

with the initialization $P(1,k) = \frac{x_i}{k} \times w_i$ (we use all remaining machines to process the last task), and $P(i,0) = +\infty$ (no solution if there are still some tasks to process but no machine left). There are $n \times m$ values of $P(i,k)$ to compute, and the computation takes a time in $O(m)$. Therefore, the complexity of this algorithm is of order $O(n \times m^2)$.

Note that it is also possible to solve the problem greedily. The idea is to assign initially one machine per task (note that there is a solution only if $m \geq n$), sort the tasks by non-increasing period, and then iteratively add a machine to the task whose machine(s) have the greater period, while there are some machines available. Let g_i be the current number of machines assigned to task T_i: the corresponding period is $\frac{x_i}{g_i} \times w_i$. At each step, we insert the task whose period has been modified in the ordered list of tasks, which can

be done in $O(\log n)$ (binary search). The initialization takes a time $O(n \log n)$ (sorting the tasks), and then there are $m - n$ steps of time $O(\log n)$. Since we assume $m \geq n$, the complexity of this algorithm is in $O(m \times \log n)$. To prove that this algorithm returns the optimal solution, let us assume that there is an optimal solution of period P_{opt} that has assigned o_i machines to task T_i, while the greedy algorithm has assigned g_i machines to this same task, and its period is $P_{greedy} > P_{opt}$. Let T_i be the task which enforces the period in the greedy solution (i.e., $P_{greedy} = x_i w_i / g_i$). The optimal solution must have given at least one more machine to this task, i.e., $o_i > g_i$, since its period is lower. This means that there is a task T_j such that $o_j < g_j$, since $\sum_{1 \leq i \leq n} o_i \leq \sum_{1 \leq i \leq n} g_i = m$ (all machines are assigned with the greedy algorithm). Then, note that since $o_j < g_j$, because of the greedy choice, $x_j w_j / o_j \geq x_i w_i / g_i$ (otherwise, the greedy algorithm would have given one more machine to task T_i). Finally, $P_{opt} \geq x_j w_j / o_j \geq x_i w_i / g_i = P_{greedy}$, which leads to a contradiction, and concludes the proof. □

Corollary 2. MinPer(spe, f_i, w_i) *can be solved in polynomial time* $O(n + m \times \log p)$.

Proof. For the specialized mapping rule, we use Corollary 1 to solve the problem: first we group the n tasks by types, therefore obtaining p equivalent tasks, in time $O(n)$. Then, we use Theorem 1 to solve the problem with p tasks, in time $O(m \times \log p)$. Finally, the computation of the mapping with equal proportions is done in $O(n)$, which concludes the proof. □

Theorem 2. MinPer$(gen, f_i, *)$ *can be solved in polynomial time.*

Proof. For the general case with $w_{i,u}$, we solve the following (rational) linear program, where the variables are P (the period), and $q(i,u)$, for $1 \leq i \leq n$ and $1 \leq u \leq m$.

$$
\begin{array}{ll}
 & \text{Minimize } P \\
 & \text{subject to} \\
\text{(i)} & q(i,u) \geq 0 \text{ for } 1 \leq i \leq n, 1 \leq u \leq m \\
\text{(ii)} & \sum_{1 \leq u \leq m} q(i,u) = x_i \text{ for each task } T_i \text{ with } 1 \leq i \leq n \\
\text{(iii)} & \sum_{1 \leq i \leq n} q(i,u) \times w_{i,u} \leq P \text{ for each machine } M_u \text{ with } 1 \leq u \leq m
\end{array}
\tag{2}
$$

The size of this linear program is clearly polynomial in the size of the instance, all $n \times m + 1$ variables are rational, and therefore it can be solved in polynomial time [21]. □

Finally, we prove that the remaining problem instances are NP-hard (one-to-many or specialized mappings, with w_u or $w_{i,u}$). Since MinPer$(o2m, f_i, w_u)$ is a special case of all other instances, it is sufficient to prove the NP-completeness of the latter problem.

Theorem 3. *The* MinPer$(o2m, f_i, w_u)$ *problem is NP-hard in the strong sense.*

Proof. We consider the following decision problem: given a period P, is there a one-to-many mapping whose period does not exceed P? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time whether it is valid or not. The NP-completeness is obtained by reduction from 3-PARTITION [12], which is NP-complete in the strong sense.

We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3n$ positive integers $a_1, a_2, \ldots, a_{3n}$ such that for all $i \in \{1, \ldots, 3n\}$, $B/4 < a_i < B/2$ and with $\sum_{i=1}^{n} a_i = nB$, does there exist a partition $I_1, \ldots, I_n$ of $\{1, \ldots, 3n\}$ such that for all $j \in \{1, \ldots, n\}$, $|I_j| = 3$ and $\sum_{i \in I_j} a_i = B$? We build the following instance $\mathcal{I}_2$ of our problem with n tasks, such that $x_i = B$ for $1 \leq i \leq n$ ($f_n = 1 - 1/B$, and $f_i = 0$ for $1 \leq i < n$). There are $m = 3n$ machines with $w_u = 1/a_u$. The period is fixed to $P = 1$. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. For $1 \leq i \leq n$, we assign task T_i to the machines of I_i: $q(i, u) = a_u$ for $u \in I_i$, and $q(i, u) = 0$ otherwise. Then, we have $\sum_{1 \leq u \leq m} q(i, u) = \sum_{u \in I_i} a_u = B$, and therefore all the work for task T_i is done. The period of machine M_u is $\sum_{1 \leq i \leq n} q(i, u) \times w_u = a_u / a_u = 1$, and therefore the period of 1 is respected. We have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. Task T_i is assigned to a set of machines, say I_i, such that $\sum_{u \in I_i} q(i, u) = B$, and $q(i, u) \leq a_u$ for all $u \in I_i$. Since all the work must be done, by summing over all tasks, we obtain $q(i, u) = a_u$, and the solution is a 3-partition, which concludes the proof. □

As a summary of the MinPer$(*, f_i, *)$ problems study, we can state that the MinPer$(*, f_i, w_i)$ problems can all be solved in polynomial time, as well as the MinPer$(gen, f_i, *)$ problems. With w_u the problem becomes NP-hard for one-to-many and specialized mappings. This also provides the complexity for the most general case of the $w_{i,u}$ problems, one-to-many and specialized mappings are NP-hard since they can be reduced to the corresponding w_u problems that are NP-hard.

4.2. Complexity of the MinPer$(*, f_{i,u}, *)$ *problems*

When we consider problems with $f_{i,u}$ instead of f_i, we do not know in advance the number of jobs to be computed by each task in order to have one job exiting the system, since it depends upon the machine on which the task is processed. However, we are still able to solve the problem with general mappings, as explained in Theorem 4. For one-to-many and specialized mappings, the problem is NP-hard with w_u, since it was already NP-hard with f_i in this case (see Theorem 3). We prove that the problem becomes NP-hard with w_i in Theorem 5, which illustrates the additional complexity of dealing with $f_{i,u}$ rather than f_i. Results are summarized in Table 1.

Theorem 4. MinPer$(gen, f_{i,u}, *)$ *can be solved in polynomial time.*

Proof. We modify the linear program (2) of Theorem 2 to solve the case with general failure rates $f_{i,u}$. Indeed, constraint (ii) is no longer valid, since the x_i

are not defined before the mapping has been decided. It is rather replaced by constraints (iia) and (iib):

$$\text{(iia)} \sum_{1 \leq u \leq m} q(n,u) \times (1 - f_{n,u}) = 1 \ ;$$

$$\text{(iib)} \sum_{1 \leq u \leq m} q(i,u) \times (1 - f_{i,u}) = \sum_{1 \leq u \leq m} q(i+1,u) \text{ for each } T_i \text{ with } 1 \leq i < n \ .$$

Constraint (iia) states that the final task must output one job, while constraint (iib) expresses the number of jobs that should be processed for task T_i, as a function of the number for task T_{i+1}. There are still $n \times m + 1$ variables which are rational, and the number of constraints remains polynomial, therefore this linear program can be solved in polynomial time [21]. □

Theorem 5. *The* MinPer$(o2m, f_{i,u}, w_i)$ *problem is NP-hard.*

Proof. We consider the following decision problem: given a period P, is there a one-to-many mapping whose period does not exceed P? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time whether it is valid or not. The NP-completeness is obtained by reduction from SUBSET-PRODUCT-EQ (SPE), which is NP-complete (trivial reduction from SUBSET-PRODUCT [12]).

We consider an instance $\mathcal{I}_1$ of SPE: given an integer B, and $2n$ positive integers $a_1, a_2, \ldots, a_{2n}$, does there exist a subset I of $\{1, \ldots, 2n\}$ such that $|I| = n$ and $\prod_{i \in I} a_i = B$? Let $C = \prod_{1 \leq i \leq 2n} a_i$.

We build the following instance $\mathcal{I}_2$ of our problem with $2n + 2$ tasks, and $2n + 2$ machines, so that the mapping has to be a one-to-one mapping (i.e., one task per machine). We ask whether we can obtain a period $P = 1$. Tasks T_1 and T_{n+2} are such that they should be allocated to machines M_{2n+1} and M_{2n+2}: the other machines never successfully compute a job for these tasks. We have:

- $w_1 = B/C^2$ and $w_{n+2} = 1/B$;
- $f_{1,2n+1} = f_{n+2,2n+2} = 0$ for $1 \leq u \leq 2n + 2$ (i.e., no failures in this case);
- $f_{1,u} = f_{n+2,v} = 1$ for $u \neq 2n + 1$ and $v \neq 2n + 2$ (i.e., total failure in this case).

The values of the failure probabilities mean that machine M_{2n+1} (resp. M_{2n+2}) never fails on task T_1 (resp. T_{n+2}), while all the other machines always fail on these tasks, i.e., the number of failed product is equal to the number of products entering the machine. No final product is output if these tasks are mapped onto such a machine.

For the other tasks ($2 \leq i \leq n + 1$ and $n + 3 \leq i \leq 2n + 2$), we have:

- $w_i = 1/C^2$ (i.e., the period is always matched);
- $f_{i,2n+1} = f_{i,2n+2} = 0$ (i.e., no failure on M_{2n+1} and M_{2n+2});
- $f_{i,u} = 1 - \frac{1}{a_u^2}$ for $2 \leq i \leq n + 1$ and $1 \leq u \leq 2n$;
- $f_{i,u} = 1 - \frac{1}{a_u}$ for $n + 3 \leq i \leq 2n + 2$ and $1 \leq u \leq 2n$.

$$\underset{T_1}{\boxed{\tfrac{B}{C^2}}} \rightarrow \underbrace{\underset{T_2}{\tfrac{1}{C^2}} \rightarrow \cdots \rightarrow \underset{T_{n+1}}{\tfrac{1}{C^2}}} \rightarrow \underset{T_{n+2}}{\boxed{\tfrac{1}{B}}} \rightarrow \underbrace{\underset{T_{n+3}}{\tfrac{1}{C^2}} \rightarrow \cdots \rightarrow \underset{T_{2n+2}}{\tfrac{1}{C^2}}}$$

Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution, I. We build the following allocation for $\mathcal{I}_2$:

- T_1 is mapped on M_{2n+1};
- T_2 is mapped on M_{2n+2};
- for $n+3 \leq i \leq 2n+2$, T_i is mapped on a machine M_u, with $u \in I$;
- for $2 \leq i \leq n+1$, T_i is mapped on a machine M_u, with $u \notin I$.

Note first that because of the values of $f_{i,u}$, the number of jobs to be computed for a task never exceeds C^2. Indeed, M_{2n+1} and M_{2n+2} never fail, and each task is mapped onto a distinct machine M_u, with $1 \leq u \leq 2n$, with a failure probability $f_u = 1 - \frac{1}{a_u^2}$ or $f'_u = 1 - \frac{1}{a_u}$. Note that $f'_u \leq f_u$ (indeed, $a_u \geq 1$ and $a_u^2 \geq a_u$), and therefore, for a task T_i, the number of products to compute is $x_i \leq \prod_{1 \leq u \leq 2n} 1/(1-f_u) = \prod_{1 \leq u \leq 2n} a_u^2 = C^2$, see Section 3.4. Therefore, the period of machines $M_1, \ldots, M_{2n}$, which are processing a task T_i with $w_i = 1/C^2$, is not greater than $x_i \times w_i = C^2 \times 1/C^2 = 1 = P$.

Now, we need to check the period of tasks T_1 and T_{n+2}. For T_{n+2}, we have $x_{n+2} = \prod_{n+2 \leq i \leq 2n+2} 1/(1 - f_{i,alloc(i)})$, where $M_{alloc(i)}$ is the machine on which T_i is mapped. Therefore, $x_{n+2} = \prod_{u \in I} a_u = B$, since I is a solution to $\mathcal{I}_1$. Since $w_{n+2,2n+2} = 1/B$, the period of machine M_{2n+2} is $B \times 1/B = 1 = P$. Finally, for T_1, $x_1 = \prod_{1 \leq i \leq 2n+2} 1/(1-f_{i,alloc(i)}) = \prod_{u \notin I} a_u^2 \times \prod_{u \in I} a_u$. We have $\prod_{u \notin I} a_u^2 = (C/B)^2$, and therefore $x_1 = C^2/B$, and the period of machine M_{2n+1} is exactly 1 as well. We have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. It has to be a one-to-one mapping, since there are no more machines than tasks. If T_1 is not mapped on M_{2n+1}, or if T_{n+2} is not mapped on M_{2n+2}, the period of the corresponding machine is at least $2P$, and hence the solution is not valid. The other tasks are therefore mapped on machines $M_1, \ldots, M_{2n}$. Let I be the set of n machines on which tasks $T_{n+3}, \ldots, T_{2n+2}$ are mapped. The period for task T_{n+2} is respected, and therefore, $\prod_{u \in I} a_u \times \frac{1}{B} \leq 1$, and

$$\prod_{u \in I} a_u \leq B \, .$$

Then, for the period of task T_1, we obtain $\prod_{u \notin I} a_u^2 \times \prod_{u \in I} a_u \times \frac{B}{C^2} \leq 1$, therefore $\prod_{u \notin I} a_u \times C \times \frac{B}{C^2} \leq 1$, and finally

$$\prod_{u \notin I} a_u \leq \frac{C}{B} \, .$$

Since $\prod_{u \in I} a_u \times \prod_{u \notin I} a_u = C$, the two inequalities above should be tight, and therefore $\prod_{u \in I} a_u = B$, which means that $\mathcal{I}_1$ has a solution. This concludes the proof. □

As a summary the complexity results for the studied MinPer problems are given in Table 1.

	f_i		$f_{i,u}$	
	w_i	w_u or $w_{i,u}$	w_i	w_u or $w_{i,u}$
o2m or spe	polynomial	NP-hard	NP-hard	NP-hard
gen	polynomial	polynomial	polynomial	polynomial

Table 1: Complexity of the MinPer problems.

In the cases where the problem is NP-complete, the global mapping problem can be split in two sub-problems: the allocation problem, i.e., how to decide the machines that are allocated to each task, and the workload balancing, i.e., how to share the tasks between these machines. The former problem can be solved in polynomial time and we provide a linear program that solves it in the next section. The first problem is more complex but we provide an integer linear program to solve simple instances in Section 4.4.

4.3. Fixed allocation of tasks to machines

If the allocation of tasks to machines is known, then we can optimally decide how to share tasks between machines, in polynomial time. We build upon the linear program of Theorem 4, and we add a set of parameters: $a_{i,u} = 1$ if T_i is allocated to M_u, and $a_{i,u} = 0$ otherwise (for $1 \leq i \leq n$ and $1 \leq u \leq m$). The variables are still the period P, and the amount of task per machine $q(i,u)$. The linear program writes:

$$
\begin{array}{lll}
\text{Minimize } P & & \\
\text{subject to} & \text{(i)} & q(i,u) \geq 0 \text{ for } 1 \leq i \leq n, 1 \leq u \leq m \\
& \text{(iia)} & \displaystyle\sum_{1 \leq u \leq m} q(n,u) \times (1 - f_{n,u}) = 1 \\
& \text{(iib)} & \displaystyle\sum_{1 \leq u \leq m} q(i,u) \times (1 - f_{i,u}) = \sum_{1 \leq u \leq m} q(i+1,u) \text{ for } 1 \leq i < n \\
& \text{(iii)} & \displaystyle\sum_{1 \leq i \leq n} q(i,u) \times w_{i,u} \leq P \text{ for } 1 \leq u \leq m \\
& \text{(iv)} & q(i,u) \leq a_{i,u} \times F_{\max} \text{ for } 1 \leq i \leq n \text{ and } 1 \leq u \leq m
\end{array}
\tag{3}
$$

We have added constraint (iv), which states that $q(i,u) = 0$ if $a_{i,u} = 0$, i.e., it enforces that the fixed allocation is respected. $F_{\max} = \prod_{1 \leq i \leq n} \max_{1 \leq u \leq m} f_{i,u}$ is an upper bound on the $q(i,u)$ values, it can be pre-computed before running the linear program. The size of this linear program is clearly polynomial in the size of the instance, all $n \times m + 1$ variables are rational, and therefore it can be solved in polynomial time [21].

4.4. Integer linear program

The linear program of Equation (3) allows us to find the solution in polynomial time, once the allocation is fixed. We also propose an integer linear program (ILP) that computes the solution to the MinPer($spe, f_{i,u}, w_{i,u}$) problem, even if the allocation is not known. However, because of the integer variables, the resolution of this program takes an exponential time. Note that this ILP can also solve the MinPer($o2m, f_{i,u}, w_{i,u}$): one just needs to assign a different type to each task.

Compared to the linear program of the previous section, we no longer have the $a_{i,u}$ parameters, and therefore we suppress constraint (iv). Rather, we introduce a set of boolean variables, $x(u,t)$, for $1 \leq u \leq m$ and $1 \leq t \leq p$, which is set to 1 if machine M_u is specialized in type t, and 0 otherwise. We then add the following constraints:

(iva) $\sum_{1 \leq t \leq p} x(u,t) \leq 1$ for each machine M_u with $1 \leq u < m$;

(ivb) $q(i,u) \leq x(u,t_i) \times F_{\max}$ for $1 \leq i \leq n$ and $1 \leq u \leq m$.

Constraint (iva) states that each machine is specialized into at most one type, while constraint (ivb) enforces that $q(i,u) = 0$ when machine M_u is not specialized in the type t_i of task T_i.

This ILP has $n \times m + 1$ rational variables, and $m \times p$ integer variables. The number of constraints is polynomial in the size of the instance. Note that this ILP can be solved for small problem instances with a solver such as ILOG CPLEX [9].

5. Heuristics

From the complexity study of Section 4, we are able to find an optimal mapping for several instances of the MinPer problem in polynomial time. General mappings are however not feasible in some cases, since it involves reconfiguring the machines between the execution of two tasks whose type is different. This additional setup time may not be affordable. So we provide in this section practical solutions to solve the MinPer($spe, f_{i,u}, w_{i,u}$) problem, which is NP-hard. As this problem is the more general problem of the MinPer set of problems its solutions can be adapted to the simpler instances.

We propose in this section a set of polynomial time heuristics that return a specialized mapping. Since we are able to find the optimal solution once the tasks are mapped onto the machines, the heuristics aims at building such an assignment. Once the assignment is computed, we run the linear program of Section 4.3 to obtain the optimal solution in terms of $q(i,u)$. Note that for all these algorithms, the memory complexity is the same, in $O(m+n)$, since we just have two tables, one for the machines and one for the tasks.

We briefly recall the notations used in the algorithms: n is the number of tasks, p is the number of task types, m is the number of machines; for $1 \leq i \leq n$ and $1 \leq u \leq m$, $t(i)$ is the type of task T_i, $w_{i,u}$ is the time taken by task T_i to

be executed on machine M_u, and $f_{i,u}$ is the failure rate for machine M_u when executing task T_i. As defined in the linear program, A_i is the set of machines on which task T_i is executed. Also, we let $UnAllocMachines$ be the set of unallocated machines, that is iteratively completed as the algorithms advance, and $bestMu$ denotes the best machine in the current loop, with an execution time $exec_b$ and a failure rate $rate_b$. When needed, $MachineLoad$ is a table with the current loads of each machine.

H1: Random heuristic. The first heuristic randomly assigns each task to a machine when the allocation respects the task type of the chosen machine. This heuristic serves as a basis for comparison and assesses the interest of providing smart solutions against random ones. For this algorithm, the time complexity is $O(mn)$: there is a loop over n tasks, and for each task, the algorithm looks for a machine that respects the task type (in the worst case, it tries all m machines).

The next three heuristics (H2, H3 and H4) are based on the same iterative allocation process in two stages. In the first *top-down* stage, the machines are assigned from task T_1 to task T_n depending on their speed $w_{i,u}$: the machine with the best $w_{1,u}$ is assigned to T_1 and so on. The motivation is that the workload of the first task is larger than the last task because of the job failures that arise along the pipeline. In the second *bottom-up* stage, the remaining machines are assigned from task T_n to task T_1 depending on their reliability $f_{i,u}$: the machine with the best $f_{n,u}$ is assigned to T_n and so on. The motivation is that it is more costly to lose a job at the end of the pipeline than at the beginning, since more execution time has been devoted to it. We iterate until all of the machines have at least one task to perform.

H2: Without any penalization. The heuristic H2 is detailed on Algorithm 1. The *top-down* stage assigns each task type to the fastest possible machine. At the end of this stage, each task of the same type is assigned onto the same machine, the fastest. Algorithm 1 loops on the task types sorted by their order in the pipeline. The assigned machines are discarded from the list. In the same way, the *bottom-up* stage assigns each task type to the same machine starting from the more reliable one but in the reverse order of the pipeline. We iterate on these two steps until all of the machines are specialized. The time complexity of the H2 algorithm is $O(m^2p)$: there are two loops on the machine numbers m, and one loop on the number of task types p.

H3: Workload penalization. The heuristic H3 is detailed on Algorithm 2. The main difference between H3 and H2 is in the execution of the *top-down* stage. In H3, each time a task is assigned to a machine, this machine is penalized to take the processing load of this task into account. So the machine's $w_{i,u}$ value is changed to $w_{i,u} \times (k+1)$, where k is the number of tasks already mapped on the machine M_u. This implies that several machines can be assigned to the same task type in this phase of the algorithm: if a machine is already loaded by several tasks, then we may find another lightly loaded machine and assign it to this task type. The *bottom-up* stage has the same behavior as

Algorithm 1: H2: Without any penalization.

```
UnAllocMachines ← {1, ..., m};
while UnAllocMachines ≠ ∅ do
    for i = 1 to p        /* top-down stage, on task types        */
    do
        exec_b ← max_u{w_{i,u}} ;
        for u ∈ UnAllocMachines do
            if w_{i,u} < exec_b then
                bestMu ← u;
                exec_b ← w_{i,u};
        A_i ← A_i ∪ {M_{bestMu}} ;
        UnAllocMachines ← UnAllocMachines \ {M_{bestMu}} ;
    for i = p to 1        /* bottom-up stage, on task types       */
    do
        rate_b ← max_u{f_{i,u}};
        for M_u ∈ UnAllocMachines do
            if f_{i,u} < rate_b then
                bestMu ← u;
                rate_b ← f_{i,u};
        A_i ← A_i ∪ {M_{bestMu}};
        UnAllocMachines ← UnAllocMachines \ {M_{bestMu}} ;
```

Algorithm 2: H3: Workload penalization.

```
UnAllocMachines ← {1, ..., m};
for u = 1 to m do
    MachineLoad[u] = 0
while UnAllocMachines ≠ ∅ do
    for i = 1 to n          /* top-down stage, on tasks          */
    do
        exec_b ← max_u{w_{i,u}} ;
        for u ∈ UnAllocMachines ∪ A_i do
            if (w_{i,u} × (MachineLoad[u] + 1)) < exec_b then
                bestMu ← u;
                exec_b ← w_{i,u};
        MachineLoad[bestMu] ← MachineLoad[bestMu] + 1;
        A_i ← A_i ∪ {M_{bestMu}} ;
        UnAllocMachines ← UnAllocMachines \ {M_{bestMu}} ;
    for i = p to 1        /* bottom-up stage, on task types      */
    do
        rate_b ← max_u{f_{i,u}} ;
        for M_u ∈ UnAllocMachines do
            if f(i, u) < rate_b then
                bestMu ← u;
                rate_b ← f_{i,u};
        A_i ← A_i ∪ {M_u} ;
        UnAllocMachines ← UnAllocMachines \ {M_u} ;
```

for H2. The time complexity of the H3 algorithm is $O(m^2n)$: the most computationally intensive part of the algorithm is the top-down stage, with two loops on the machine numbers m, and one loop on the number of tasks n.

H4: Cooperation work. The heuristic H4 is detailed on Algorithm 3. In this heuristic, during the *top-down* stage, a new machine is assigned to each task. The fastest machines are assigned first. So after this stage the assigned machines are not shared between tasks. Note however that the linear program used to optimize the load balance after the assignment stage will redistribute the tasks on the machines afterwards. Then the *bottom-up* stage has the same behavior as the heuristic H2. The time complexity of the H4 algorithm is $O(m^2n)$: the most computationally intensive part of the algorithm is the top-down stage, with two loops on the machine numbers m, and one loop on the number of tasks n.

H5: Focus on speed. Finally, H5 performs only a *top-down* stage, repetitively. The heuristic H5 is detailed on Algorithm 4. It focuses only on the speed by repeating the *top-down* stage previously presented for the H3 heuristic until all the machines are allocated to at least one task. The time complexity

Algorithm 3: H4: Cooperation work.

```
UnAllocMachines ← {1, ..., m};
while UnAllocMachines ≠ ∅ do
    for i = 1 to n          /* top-down stage, on tasks          */
    do
        exec_b ← max_u{w_{i,u}} ;
        for u ∈ UnAllocMachines do
            if w_{i,u} < exec_b then
                bestMu ← u;
                exec_b ← w_{i,u};
        A_i ← A_i ∪ {M_{bestMu}} ;
        UnAllocMachines ← UnAllocMachines \ {M_{bestMu}} ;
    for i = p to 1          /* bottom-up stage, on task types    */
    do
        rate_b ← max_u{f_{i,u}} ;
        for M_u ∈ UnAllocMachines do
            if f_{i,u} < rate_b then
                bestMu ← u;
                rate_b ← f_{i,u};
        A_i ← A_i ∪ {M_{bestMu}};
        UnAllocMachines ← UnAllocMachines \ {M_{bestMu}} ;
```

of the H5 algorithm is $O(m^2n)$: the most computationally intensive part of the algorithm is the top-down stage, with two loops on the machine numbers m, and one loop on the number of tasks n.

Algorithm 4: H5: Focus on speed.

```
UnAllocMachines ← {1, ..., m};
for u = 1 to m do
    MachineLoad[u] = 0
while UnAllocMachines ≠ ∅ do
    for i = 1 to n            /* top-down stage on tasks          */
    do
        exec_b ← max_u{w_{i,u}} ;
        for u ∈ UnAllocMachines ∪ A_i do
            if (w_{i,u} × (MachineLoad[u] + 1)) < exec_b then
                bestMu ← u;
                exec_b ← w_{i,u};
        MachineLoad[bestMu] ← MachineLoad[bestMu] + 1;
        A_i ← A_i ∪ {M_{bestMu}} ;
        UnAllocMachines ← UnAllocMachines \ {M_{bestMu}} ;
```

6. Simulation results

In this section, we assess the performance of the five heuristics. Note that there is no need to use or implement a simulator since the task allocations can directly be computed by implementing the different heuristics in simple programs.

The period returned by each heuristic is measured in *ms* as the $w_{i,u}$ values used by the programs are given in *ms*. Recall that m is the number of machines, p the number of types, and n the number of tasks. Each point in a figure is an average value of 30 simulations, where the $w_{i,u}$ are randomly chosen between 100 and 1000 *ms*, for $1 \leq i \leq n$ and $1 \leq u \leq m$, unless stated otherwise. Similarly, failure rates $f_{i,u}$ are randomly chosen between 0.2 and 10 % (i.e., 1/500 and 1/10), unless stated otherwise.

6.1. Heuristic results compared to optimal solutions

In this set of simulations, the heuristics are compared to the integer linear program that gives the optimal solution. The platform is such that $m = 20$, $p = 5$ and $21 \leq n \leq 61$. Figure 6 shows that the random heuristic H1 has poor performance and so that not every simple algorithm could have reasonable performance. This poor result is not limited to this particular simulation and therefore, for visibility reasons, H1 does not appear in the remaining figures.

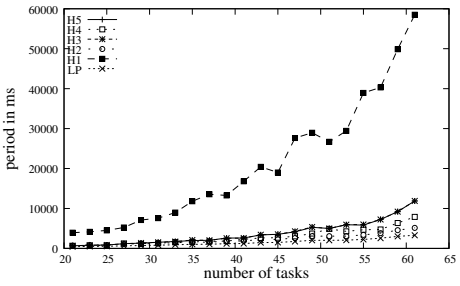

Figure 6: Heuristics compared to the ILP.

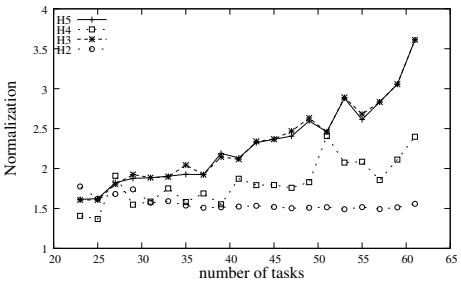

Figure 7: Normalized results without H1.

We focus in Figure 7 on the heuristics H2 to H5. The presented results are normalized upon the linear program results. Most of the heuristics have performance with a factor between 1.5 and 3.5 from the optimal solution and the difference increases with the number of tasks. The algorithm behaviors are moreover rather unstable, except for H2. This probably means that there is few guarantee on the quality of the found solutions. This instability comes from the low values taken for m and p. These values has been chosen because in this configuration, although exponential, the linear program always finds a result. For a platform with 20 machines and 10 types instead of 5, the percentage of success of the linear program collapses to less than 50% for 61 tasks and the comparison between the algorithms and the ILP is not meaningful.

In the remaining simulations, we concentrate on higher values of m and p to limit the algorithms instability. We are however not able to compute optimal results for these configurations.

6.2. General behavior of the heuristics

In a second set of simulations, we focus on the difference between having more tasks than machines or the opposite. In Figure 8, we use platforms with 50 machines and 25 types of tasks and the number of tasks varies between 10 and 50. The results show that H2 is slightly less performing than the other heuristics while it was the best in the previous simulations. This difference increases as the number of types gets closer to the number of machines, as shown in Figure 9.

When the number of tasks is higher than the number of machines, H2 and H4 become clearly the best heuristics (see Figure 10). Indeed, at the end of the first allocation stage, H3 and H5 will almost have used all of the machines and the second stage will thus not be decisive. This is why the lines of H3 and H5 are superimposed in this case.

To study the impact of the speed of the machines, we set a platform with almost homogeneous machines ($100 \leq w_{i,u} \leq 200$). Results are presented in Figure 11 and shows that the homogeneity in terms of the machine speed does not change the overall behavior of the heuristics. H2 and H4 are still the best even if the gap with H3 and H5 is reduced.

To conclude with general behavior, we studied the impact of the number of types with two sets of platforms with 40 machines and a number of tasks ranging

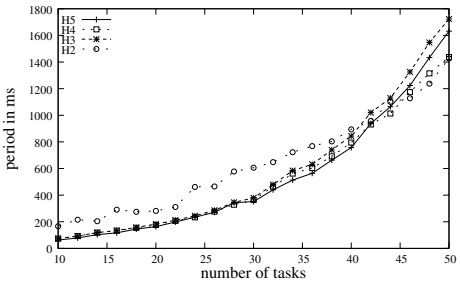

Figure 8: $m = 50$, $p = 25$.

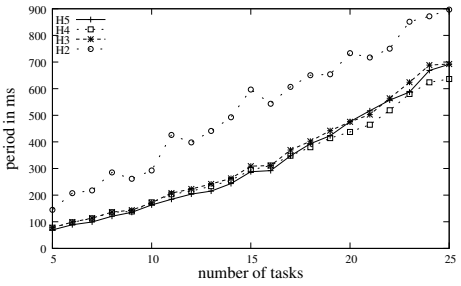

Figure 9: $m = 25$, $p = 15$.

Heuristics with more machines than tasks.

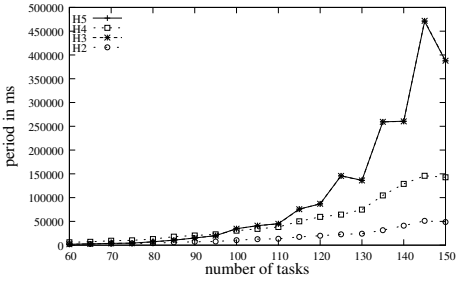

Figure 10: $m = 50$, $p = 25$.
Heuristics with more tasks than machines.

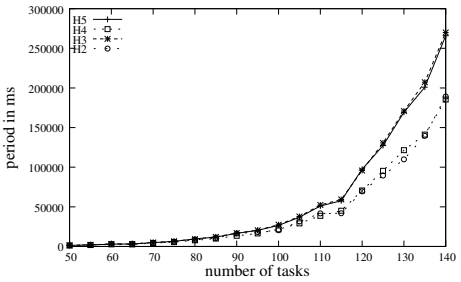

Figure 11: $m = 40$, $p = 30$, $100 \leq w_{i,u} \leq 200$.
Homogeneous machines.

from 10 to 110. The number of task types is set to 5 for the first one (Figure 12) and to 35 for the second one (Figure 13). When the number of types is small compared to the number of machines (Figure 12), the opportunity to split groups is high. In this case, H2 and H4 are the best heuristics because the workload is shared among a bigger set of machines and not only on those that are efficient for a given task. On the opposite, when the number of types is close to the number of machines, the number of split tasks decreases. Indeed, each machine must be specialized to one type. In the simulation shown in Figure 13, only 5 machines can be used to share the workload once each machine is dedicated to a type. That explains why the performance of the heuristics are pretty much alike.

6.3. Impact of the failure rate

In this last set of simulations, we study the impact of the failure rate on the heuristics. Figures 14 and 15 show that when the failure rate is high ($0 \leq f_{i,u} \leq 30\%$), only H2 and H4 have a good performance. Remember that the heuristics have two stages, the first one optimizes the $w_{i,u}$ and the second one the $f_{i,u}$. In the case of H3 and H5, the first stage does not encourage the reuse

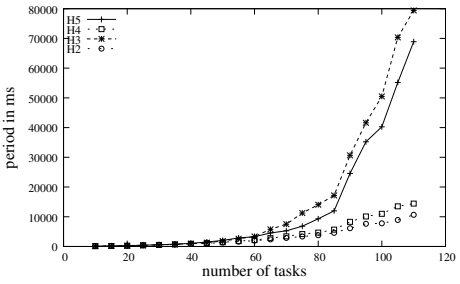

Figure 12: $m = 40$, $p = 5$.
Small number of types.

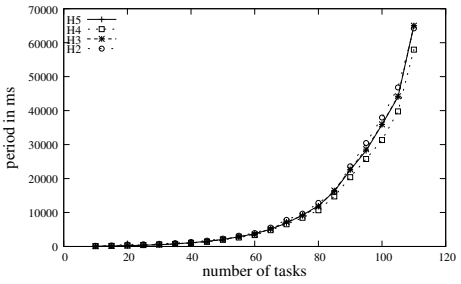

Figure 13: $m = 40$, $p = 35$.
High number of types.

of a machine already assigned to a task (the penalization is high). Thus, in the particular case of the platform set in Figures 14 and 15, H3 and H5 assign all the machines at the end of their first stage and cannot optimize the failure in the second stage because no more machine is available.

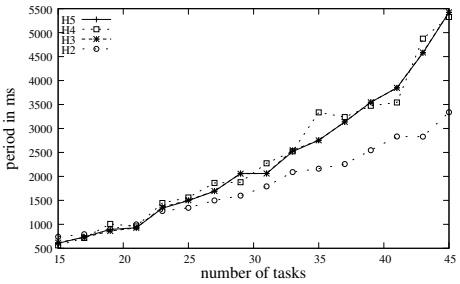

Figure 14: $m = 15$, $p = 5$.
Failure $0 \leq f_{i,u} \leq 10\%$.

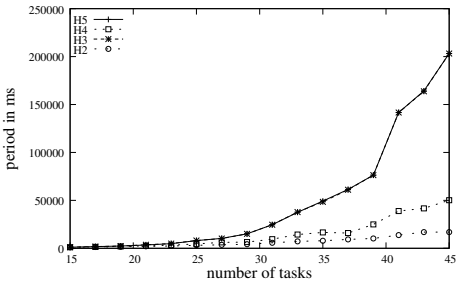

Figure 15: $m = 15$, $p = 5$.
Failure $0 \leq f_{i,u} \leq 30\%$.

6.4. Detailed results per heuristic

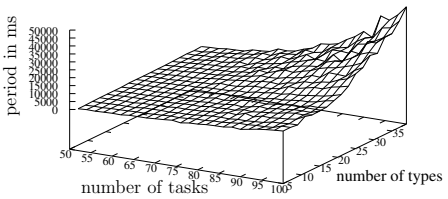

Figure 16: Heuristic H2.

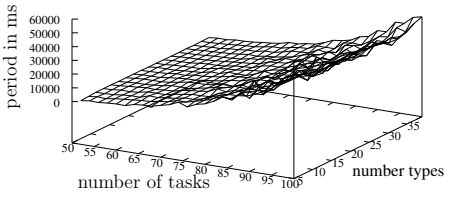

Figure 17: Heuristic H3.

To give a wider view of the algorithm behaviors depending on the number of tasks and on the number of types, we provide 3D curves for H2 to H5. The presented results are computed for $m = 40$. We can note that until 60 tasks the algorithms are equivalent. Then if the number of tasks increases, H2 always gives

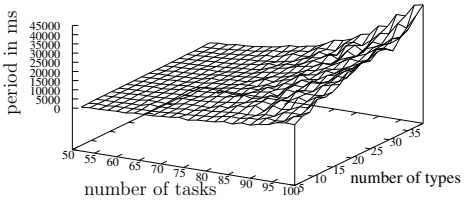

Figure 18: Heuristic H4.

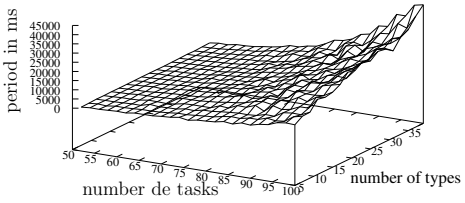

Figure 19: Heuristic H5.

a shorter period unless the number of types is bigger than 25. H3 outperforms H2 when the number of types is high. One of the reasons of this better behavior may be that this algorithm uses a lower number of machines in the first stage. So it leaves more machines to be assigned on the failure rate criterion and thus achieves a better balance between speed and reliability. This remark is enforced if we note that the performance of H2 deteriorates when the number of types increases, leading to more machine assignments in the first case.

7. Conclusion

In this paper, we investigate the problem of maximizing the throughput of coarse-grain pipeline applications where tasks have a type and are subject to failures. We propose mapping strategies to solve the problem considering three rules of the game: one-to-many mappings (each machine processes at most one task), specialized mappings (several tasks of the same type per machine), or general mappings. In any case, the jobs associated to a task can be distributed upon the platform so as to balance workload between the machines. From a theoretical point of view, an exhaustive complexity study is proposed. We prove that an optimal solution can be computed in polynomial time in the case of general mappings whatever the application/platform parameters, and in the case of one-to-many and specialized mappings when the faults only depend on the tasks, while the optimization problem becomes NP-hard in any other cases. Since general mappings do not provide a realistic solution because of not affordable setup times when reconfiguration occurs, we propose to solve the specialized mapping problem by designing several polynomial heuristics. Also, we give an integer linear programming formulation of the problem that allows us to compute an optimal solution on small problem instances and to evaluate the performance of these heuristics on such instances. The simulations show that some heuristics return specialized mappings with a throughput close to the optimal, and that using random mappings never gives good solutions. As future work, we plan to investigate other objective functions, as the latency, or other failure models in which the failure rate associated to the task and/or the machine is correlated with the time to perform that task.

Acknowledgment

Anne Benoit is with Institut Universitaire de France. This work was supported in part by the ANR RESCUE project.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[2] A. Benoit, U. V. Çatalyurek, Y. Robert, and E. Saule. A Survey of Pipelined Workflow Scheduling: Models and Algorithms. Technical Report RR-LIP-2010-28, LIP, ENS Lyon, France, Sept. 2010.

[3] A. Benoit, A. Dobrila, J.-M. Nicod, and L. Philippe. Throughput optimization for micro-factories subject to failures. In *Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing*, ISPDC '09, pages 11–18, Washington, DC, USA, 2009. IEEE Computer Society.

[4] A. Benoit, V. Rehn-Sonigo, and Y. Robert. Optimizing latency and reliability of pipeline workflow applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –10, april 2008.

[5] A. Benoit, V. Rehn-Sonigo, and Y. Robert. Optimizing Latency and Reliability of Pipeline Workflow Applications. In *HCW'08, the 17th Heterogeneity in Computing Workshop*. IEEE Computer Society Press, 2008.

[6] J. Blażewicz, M. Drabowski, and J. Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. Comput.*, 35:389–393, May 1986.

[7] J. Burge, K. Munagala, and U. Srivastava. Ordering pipelined query operators with precedence constraints. Technical Report 2005-40, Stanford InfoLab, 2005.

[8] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Góes, and W. Voorsluys. On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Computing*, 33(3):213–234, 2007.

[9] ILOG CPLEX: High-performance software for mathematical programming and optimization. `http://www.ilog.com/products/cplex/`.

[10] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005.

[11] A. Deshpande and L. Hellerstein. Parallel pipelined filter ordering with precedence constraints. *ACM Transactions on Algorithms*, 2010.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[13] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34:151–162, October 2006.

[14] H. Gröflin, A. Klinkert, and N. P. Dinh. Feasible job insertions in the multiprocessor-task job shop. *European J. of Operational Research*, 185(3):1308 – 1318, 2008.

[15] F. Guirado, A. Ripoll, C. Roig, A. Hernàndez, and E. Luque. Exploiting throughput for pipeline execution in streaming image processing applications. In *Proceedings of the 12th international conference on Parallel Processing*, Euro-Par'06, pages 1095–1105, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] T. D. R. Hartley, A. Fasih, C. A. Berdanier, F. Özgüner, and Ü. V. Çatalyürek. Investigating the use of gpu-accelerated nodes for sar image formation. In *CLUSTER*, pages 1–8, 2009.

[17] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[18] A. Litke, D. Skoutas, K. Tserpes, and T. Varvarigou. Efficient task replication and management for adaptive fault tolerance in mobile grid environments. *Future Generation Computer Systems*, 23(2):163 – 178, 2007.

[19] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 574–579, New York, NY, USA, 2008. ACM.

[20] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. Elastic scaling of data parallel operators in stream processing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[21] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer-Verlag, 2003.

[22] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In T. Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[23] J. B. Weissman and D. Womack. Fault tolerant scheduling in distributed networks. Technical Report TR CS-96-10, Department of Computer Science, University of Texas, San Antonio, Sept. 1996.

[24] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *Proc. of the 21st IEEE Real-Time Systems Symp.*, pages 239–248. IEEE, 2000.

[25] R. West and K. Schwan. Dynamic Window-Constrained Scheduling for Multimedia Applications. In *ICMCS, Vol. 2*, pages 87–91, 1999.

[26] R. West, Y. Zhang, K. Schwan, and C. Poellabauer. Dynamic window-constrained scheduling of real-time streams in media servers. *IEEE Transactions on Computers*, 53:744–759, 2004.

[27] M. Wieczorek, A. Hoheisel, and R. Prodan. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Gener. Comput. Syst.*, 25(3):237–256, 2009.

[28] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. Research Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, Apr. 2005.